Managing Medical Research Data with a Web-Interfacing Repository Manager

R.M. Jakobovits and J.F. Brinkley
Departments of Computer Science and Biological Structure
University of Washington, Seattle, WA

This paper describes the Web-Interfacing Repository Manager (WIRM), a perl toolkit for managing and deploying multimedia data, which is built entirely from free, platform-independent components. The WIRM consists of an object-relational API layered over a relational database, with built-in support for file management and CGI programming. The basic underlying data structure for all WIRM data is the repository object, *a perl associative array whose values are bound to a row of a table in the relational database. Based on our experience implementing a target application (the* Brain Mapper Console*), we describe five stages through which a system passes as it evolves from a primitive file hierarchy to a full-fledged repository console.*

INTRODUCTION

Medical research tends to be characterized by experiments which produce large collections of multimedia files [1]. These data files may consist of a wide range of image types, custom-formatted binary data, and ASCII dumps of alphanumeric tables. For example, functional brain mapping data consists of thousands of ordered MRI slices grouped into exams, 3-D rendered brain images, digitized intra-operative photographs, lists of identified site coordinates, and alphanumeric tables of patient demographics [2]. In addition to the actual data files, experiment management involves handling a wide range of ancillary file objects that aren't traditionally considered to be "data", such as the procedural scripts used to generate the data, descriptive HTML and word processing documents, and the all-important *README* text files scattered about the directory hierarchy, which are often the only recorded source of crucial metadata about files in the directories. These ancillary files are often numerous and hard to manage, and can themselves be considered "multimedia data" from a file management perspective.

Without the proper tools, file management may gradually become an overwhelming task as the amount of file-based information increases. We have identified four major requirements that need to be addressed when handling medical multimedia: metadata management, query support, user interface construction, and application interfacing.

Each of these requirements can be addressed by a number of popular technologies. Metadata management can be aided by enforcing strict directory maintenance through a revision control system [3]. Queries can be supported by relegating tabular data to a relational database. User interfaces can be constructed as CGI or Java applications. Files can be interfaced to applications by perl scripts, the language of choice for managing processes and handling files [4].

A number of competing off-the-shelf products attempt to provide more complete solutions by integrating these features within a single tool. The drawbacks of these commercial systems are that they require a significant investment in monetary and personnel resources. In addition to the high price tag of the software itself (often costing thousands of dollars), you may need to invest in new hardware and personnel training. Furthermore, when your repository is tied into a proprietary vendor's system, you can not freely distribute your results within the research community.

This paper describes the Web-Interfacing Repository Manager (WIRM), a perl toolkit for managing and deploying multimedia data, which is built entirely from free, platform-independent components. In a repository system, metadata about each file object are maintained in a database, and access to all data is regulated by a layer of control services called a *repository manager*[5]. The WIRM is essentially an object-relational API [6] layered over a relational database (MiniSQL) [7], with built-in support for file management and CGI programming. The basic underlying data structure for all WIRM data is the ***repository object***, which is a perl associative array whose values are bound to a row of a table in the relational database. The WIRM architecture, shown in Figure 1, is described in in greater detail in [8].

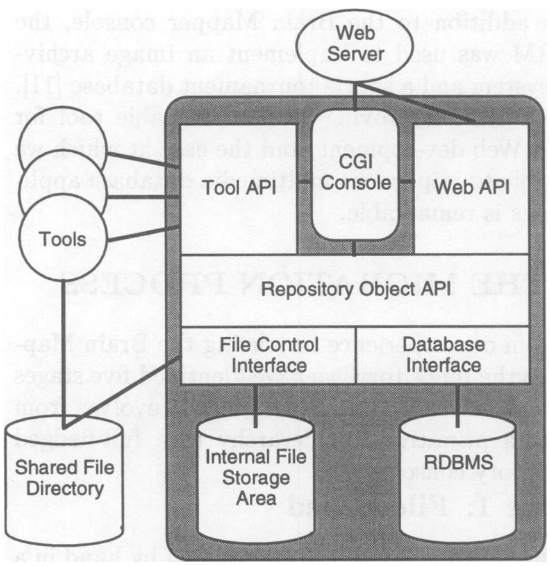

Figure 1: The WIRM Architecture

WIRM Architecture

The WIRM Data Model

The Repository Object API provides an object-relational data model to the CGI programmer by abstracting away the relational tables and allowing the data to be viewed as collections of objects. Each object has a ***schema*** which determines it's structure, and a unique *Object Identifier (OID)*, by which it can be efficiently referenced from other objects. A repository object can be an atomic type (string, integer, real, OID), a file (with associated metadata), an aggregate type (set or list), or a composite user-defined type.

Each composite type is implemented as a table in the relational database, with a column for every attribute, plus an extra column to hold the OID. When an instance of the type is created, a row is added to the table, and an associative array is allocated whose keys correspond to the column names, and whose values are bound to the row data. The Repository Object API defines query functions which utilize the relational query engine to retrieve rows which satisfy an SQL query, and then allocates an associative array for each row. The perl programmer may iterate over the results, performing arbitrary computations, and updates are propagated back to the table, where they persist between invocations. Just as an object-oriented database can be described as ***persistent C++***, the WIRM can be described as ***persistent perl***, with the added benefit of a relational query engine and built-in multimedia support.

The Web API

The Web API is a toolkit for the rapid building of Web-based consoles. It provides a suite of functions for creating and parsing form elements (e.g. popup menus, scrollable lists, etc.), based on the free perl module CGI.pm [9]. It also provides many shortcuts for generating HTML syntax (e.g. turning a perl array into a formatted table, handling document layout, displaying a thumbnail image, etc.), and a high-level interface for displaying query results. The API supports two distinct abstractions for query display: the statement handle visualization methods, which interface directly to the relational DB API, and the repository object visualization methods, which operate on perl associative arrays.

In addition, the Web API provides invaluable utility functions such as *get_url*, which retrieves the contents of a specified remote url into a string, and *mime_type*, which returns the type of a file based on it's extension.

The File Control Interface

The File Control Interface regulates access to the File Storage Area (FSA). In the original version of the WIRM, files checked out of the FSA were copied to a shared directory, but for the sake of efficiency, read-only requests are now granted direct access to the FSA. New files are imported to the FSA by calls to the *file_import* function, which stores a file and it's associated metadata in the repository.

The DB API

The DB API contains functions for issuing SQL select, join, update, and delete statements, and for managing database connections and logging. The select statements return ***statement handles***, which

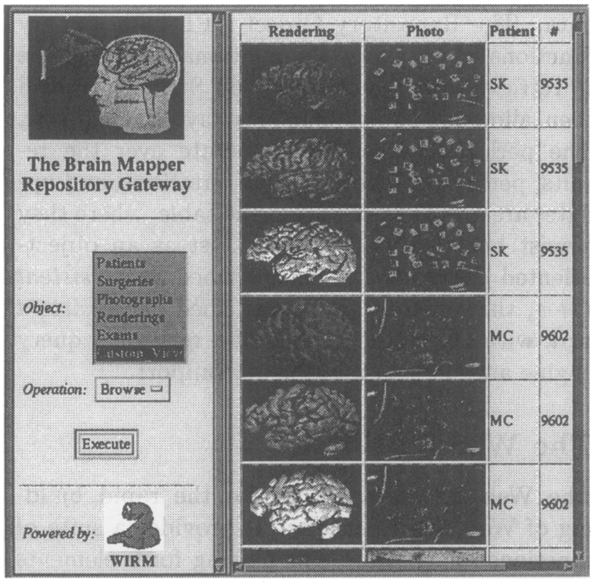

Figure 2: CGI console built using WIRM

are 2D arrays of table data. The API uses the free Msql-Perl Adaptor [10].

A TARGET APPLICATION

The WIRM has been used to implement the Brain Mapper console, including graphical data acquisition forms, scripts for importing existing file-based legacy data, a patient-centered browser for viewing photographs and renderings, a generic gateway for posing ad-hoc queries across all data types, and an interface for connecting to a remote image server to upload MRI slices. The console, shown in Figure 2, currently maintains multimedia data for twelve patients, including demographics and surgery information, MRI slices, radiology exam parameters, digitized intra-operative photographs, 3-D surface and volume renderings, stimulation studies, and spatial mapping data from multiple authors.

As evidence of the high-level programming efficiency of the WIRM Web and Repository Object API's, the entire Brain Mapper console is only 330 lines of perl code. Because of the layered nature of the WIRM architecture, there is a high level of code reuse, and the API's themselves are extremely compact. The Web API and Tool API leverage highly off the Repository Object API, which in turn makes good use of the DB-API and the File-API. Together, the five WIRM API's consist of only 1100 lines of perl code.

The development of the WIRM API's proceeded in parallel with the implementation of it's driving application, the Brain Mapping console. Wherever appropriate, we generalized solutions from the Brain Mapper to be part of the WIRM system. For example, the Brain Mapper needed to display an HTML table of thumbnail images which could be clicked to view the full-sized images, and this function was added to the Web API, where it could be used in other applications.

In addition to the Brain Mapper console, the WIRM was used to implement an image archiving system and a sports tournament database [11]. The WIRM is proving to be a valuable tool for rapid Web development, and the ease at which we are able to implement multimedia database applications is remarkable.

THE MIGRATION PROCESS

From our experience in porting the Brain Mapper to the repository, we have identified five stages through which a system passes as it evolves from being a primitive file hierarchy to a full-fledged repository console.

Stage 1: File-Based

In this stage, all data are maintained by hand in a directory hierarchy. Metadata are implicitly embedded in each file's name and location, or scattered across the directory tree in hand-written text files. For example, consider the Brain Mapping system before the repository was introduced. For each patient, the MRI exams, photographs, and map files were all stored in a directory named after that patient. A file *kevin.map* located in a directory *JSP* implied that the file contained map data about patient JSP's brain, authored by Kevin. No queries were supported, and users could only interact with the data by browsing the file system and launching external applications (such as image viewers) on the desired files by hand.

Stage 2: BLOBs

In the second stage, the basic file objects were modeled as simple repository schemata. For example, a repository object schema containing three attributes (Patient, Author, Data_File) was defined to represent the maps described above. A perl script was written to traverse the directory hierarchy looking for files with the ".map" extension. When the script encountered a map file, it copied the file into the repository's FSA using the *file_import* function, and created a new Map object using the *repo_new* function. In this way, the metadata about the map file were rendered explicitly in the repository, and could be queried via the repository query mechanism. Using the WIRM Web API, it was trivial to implement a graphical CGI console which supported retrieval of map files by selecting from menus of patients and authors, requiring only 5 lines of perl code.

At this stage, the repository knows nothing about the actual coordinate data within the map file. The map files are treated as *Binary Large Objects (BLOBs)*, and must be processed by external applications at the file system level.

Stage 3: Schema Evolution

To enable the repository to support queries involving the map coordinate data, the contents of the BLOBs must be revealed to the query mechanism. This *deblobification* process requires the database designer to have an in-depth understanding of the inherent structure of the data to be modeled, and to evolve the schema to incorporate this structure. This evolution is an ongoing process, and the degree to which a multimedia database is deblobified tends to increase over time as the system matures. To support this process, we implemented a *schema_evolve* facility as part of the Repository Object API, which allows the database designer to edit the structure of a schema type, and have the changes propagate to all existing instances of that type. For example, the Map schema defined in stage 2 was evolved to include a fourth attribute: a *List of Sites*. A *Site* is a new object type, which consists of a *Site-Label* and a triplet of spatial coordinates. Then a script was written to parse every map data file and populate the repository with Site objects. Once this was done, the query system could be used to answer sophisticated ad-hoc queries such as:

- "which sites on patient JSP have a discrepancy of more than 1 cm between Kevin's map and Jim's map?"
- "display the photos of patients who have more than 3 sites in their POB region"
- "Do patients with higher verbal IQ's have more language sites?"

Stage 4: Critical Mass

Until this stage, the repository is essentially a copy of the original file hierarchy, with the added features of query support and data display. If the repository becomes corrupted or stale, it can easily be refreshed by dropping the entire database and file storage area, and executing the import scripts to repopulate the repository. This "bootstrapping" process is only possible while the original file hierarchy is a complete image of the data, but at some point the repository will be used as a data acquisition tool to collect new data. The Web API makes it easy to construct Web forms for uploading files and alpha-numeric data directly into the repository, and such data will not be mirrored in the original file hierarchy. When this happens, a *critical mass point* is reached from which there is no going back: the repository can no longer be boot-strapped without losing data. Although the threat of data loss can be assuaged by using standard transactional logging, checkpoint, and recovery techniques, the user might fear "relinquishing the reigns" from their familiar file hierarchy and putting their precious data completely at the mercy of the WIRM. This can be especially disconcerting when viewing the repository's file storage area, where the files have been assigned mysterious numeric filenames and reside in a cryptic directory hierarchy that bears no resemblance to the original file organization. All the implicit, human-readable metadata has been stripped from the directory structure and relegated to the database. Without some redundancy in the form of human-readable file organization, the files are effectively lost if the database happens to fail. To placate the users, we split the file storage area into two partitions: the *default area*, in which files are assigned names according to their Object ID and hashed into directories for efficient lookup, and the *custom area*, which contains a user-specified, human readable file hierarchy. We augmented the *file_import* function to allow users to circumvent the default destination by supplying their own path and filename. We are hoping that the Brain Mapper users will soon be convinced that their data are safe in the FSA, at which point we can free up the gigabytes of disk space devoted to maintaining the original file hierarchy.

Stage 5: Tool Integration

In the final stage, the focus shifts to integration of external software applications, which are viewed as *tools* from the repository's perspective. The Web interface which was developed for data display and acquisition can now be extended to support data processing, acting as a launch pad for tools. In the Brain Mapper, we have integrated the Web console with a tool that fetches MRI exams from a remote image server. We are still in the early stages of tool integration, and the Tool API needs to be further developed to support a range of generic operations between repository objects and applications which process them.

FUTURE WORK

In addition to further developing the Tool API, we are considering adding the following features:

- a more sophisticated transaction model, in which updates are performed automatically when a transaction commits, rather than requiring the programmer to explicitly update the objects.
- better support for aggregate types at the Repository Object API level.
- making tables orthogonal to type, as suggested by Stonebraker[6].

The WIRM has evolved from being a custom tool for the Brain Mapper project, to being a general tool for supporting a wide range of multimedia applications. Now that the API definitions have stabilized, we plan to package the WIRM as a portable perl module, develop a user manual and tutorial, and release it for beta testing on the Web. The UW Digital Anatomist Home Page [12] will have information regarding the release.

Acknowledgments

This work was funded by Human Brain project grant DC/LM02310, co-funded by the National Institute for Deafness and Other Communication Disorders, and the National Library of Medicine.

References

1. L. Shapiro, S. Tanimoto, J. Brinkley, J. Ahrens, R. Jakobovits, and L. Lewis. A visual database system for data and experiment management in model-based computer vision. In *Proceedings of the Second CAD-Based Vision Workshop*, pages 64–72, February 1994.
2. B. R. Modayur, J. Prothero, C. Rosse, R. Jakobovits, and J.F. Brinkley. Visualization and mapping of neurosurgical functional brain data onto a 3-D MR-based model of the brain surface. In *AMIA Fall Symposium*, pages 304–308, 1996.
3. W. F. Ticy. RCS - a system for version control. *IEEE Software Practice and Experience*, 15(7):637–654, 1985.
4. L. Wall and R.L. Schwartz. *Programming perl.* O'Reilly & Associates, Inc., Sebastopol, CA, 1991.
5. P.A. Bernstein and U. Dayal. An overview of repository technology. In *Proceedings of the 20th VLDB Conference*, September 1994.
6. M. Stonebraker. *Object-Relational DBMSs: The Next Great Wave.* Morgan Kaufmann, 1996.
7. D. Hugues. Mini SQL: A lightweight database server. http://bond.edu.au/People/bambi/-mSQL/.
8. R. Jakobovits, B. Modayur, and J.F. Brinkley. A web-based repository manager for brain mapping data. In *AMIA Fall Symposium*, pages 309–314, 1996.
9. L. Stein. CGI.pm - a perl5 CGI library. http://www-genome.wi.mit.edu/ftp/pub-/software/www/cgi_docs.html.
10. A. Koenig. The Msql perl adaptor. ftp://Bond.edu.au/pub/-Minerva/msql/Contrib/MsqlPerl.README.
11. The university of washington internet racquetball ladder. http://www4.biostr.washington.edu/UWIRL.
12. The digital anatomist home page. http://-www1.biostr.washington.edu/-DigitalAnatomist.html.